Sequential Logic Optimization

- **State Minimization**
  - Algorithms for State Minimization
- **State, Input, and Output Encodings**
  - Minimize the Next State and Output logic

Optimization in Context

- Understand the word specification
- Draw a picture
- Derive a state diagram and Symbolic State Table
- Determine an implementation approach (e.g., gate logic, ROM, FPGA, etc.)
- Perform **STATE MINIMIZATION**
- Perform **STATE ASSIGNMENT**
- Map Symbolic State Table to Encoded State Tables for implementation (**INPUT and OUTPUT encodings**)  
- You can specify a specific state assignment in your Verilog code through parameter settings
Finite State Machine Optimization

- **State Minimization**
  - Fewer states require fewer state bits
  - Fewer bits require fewer logic equations

- **Encodings: State, Inputs, Outputs**
  - State encoding with fewer bits has fewer equations to implement
    - However, each may be more complex
  - State encoding with more bits (e.g., one-hot) has simpler equations
    - Complexity directly related to complexity of state diagram
  - Input/output encoding may or may not be under designer control

Algorithmic Approach to State Minimization

- **Goal** - identify and combine states that have equivalent behavior
- **Equivalent States:**
  - Same output
  - For all input combinations, states transition to same or equivalent states

- **Algorithm Sketch**
  - 1. Place all states in one set
  - 2. Initially partition set based on output behavior
  - 3. Successively partition resulting subsets based on next state transitions
  - 4. Repeat (3) until no further partitioning is required
    - States left in the same set are equivalent
  - Polynomial time procedure
State Minimization Example

Sequence Detector for 010 or 110

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset 0</td>
<td>S0</td>
<td>S1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S1</td>
<td>S3</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>S2</td>
<td>S5</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>S4</td>
<td>S0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>S5</td>
<td>S0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>S6</td>
<td>S0</td>
<td>1</td>
</tr>
</tbody>
</table>

Method of Successive Partitions

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset 0</td>
<td>S0</td>
<td>S1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S1</td>
<td>S3</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>S2</td>
<td>S5</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>S4</td>
<td>S0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>S5</td>
<td>S0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>S6</td>
<td>S0</td>
<td>1</td>
</tr>
</tbody>
</table>

( S0 S1 S2 S3 S4 S5 S6 )
S1 is equivalent to S2
( S0 S1 S2 S3 S5 ) ( S4 S6 )
S3 is equivalent to S5
( S0 S1 S2 ) ( S3 S5 ) ( S4 S6 )
S4 is equivalent to S6
( S0 ) ( S1 S2 ) ( S3 S5 ) ( S4 S6 )
Minimized FSM

- State minimized sequence detector for 010 or 110

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State X=0</th>
<th>Next State X=1</th>
<th>Output X=0</th>
<th>Output X=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset 0 + 1</td>
<td>S0</td>
<td>S1'</td>
<td>S1'</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X0</td>
<td>S3'</td>
<td>S0</td>
<td>S0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X1</td>
<td>S4'</td>
<td>S0</td>
<td>S0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

7 States reduced to 4 States
3 bit encoding replaced by 2 bit encoding

Another Example

- 4-Bit Sequence Detector: output 1 after each 4-bit input sequence consisting of the binary strings 0110 or 1010
# State Transition Table

- **Group states with same next state and same outputs**

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State $X=0$</th>
<th>Output $X=0$</th>
<th>Output $X=1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>$S_0$</td>
<td>$S_1$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$S_1$</td>
<td>$S_4$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$S_2$</td>
<td>$S_5$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>$S_3$</td>
<td>$S_7$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$S_4$</td>
<td>$S_0$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>$S_5$</td>
<td>$S_8$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>$S_6$</td>
<td>$S_{10}$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>$S_7$</td>
<td>$S_{11}$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>$S_8$</td>
<td>$S_{12}$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>$S_9$</td>
<td>$S_0$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>$S_{10}$</td>
<td>$S_0$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>$S_{11}$</td>
<td>$S_0$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>011</td>
<td>$S_{12}$</td>
<td>$S_0$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>$S_{13}$</td>
<td>$S_0$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>$S_{14}$</td>
<td>$S_0$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>$S_0$</td>
<td>$S_0$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>$S_0$</td>
<td>$S_0$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

---

# Iterate the Row Matching Algorithm

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State $X=0$</th>
<th>Output $X=0$</th>
<th>Output $X=1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>$S_0$</td>
<td>$S_1$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$S_1$</td>
<td>$S_4$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$S_2$</td>
<td>$S_5$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>$S_3$</td>
<td>$S_7$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$S_4$</td>
<td>$S_0$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>$S_5$</td>
<td>$S_8$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>$S_6$</td>
<td>$S_{10}$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>$S_7$</td>
<td>$S_{11}$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>$S_8$</td>
<td>$S_{13}$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>$S_9$</td>
<td>$S_0$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>$S_{10}$</td>
<td>$S_0$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>$S_{11}$</td>
<td>$S_0$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S'$_7$</td>
<td>011 or 101</td>
<td>$S_{10}$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$S_{12}$</td>
<td>$S_0$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>$S_{13}$</td>
<td>$S_0$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>$S_{14}$</td>
<td>$S_0$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>$S_0$</td>
<td>$S_0$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Iterate One Last Time

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$X=0$</td>
<td>$X=1$</td>
</tr>
<tr>
<td>Reset</td>
<td>$S_0$</td>
<td>$S_1$</td>
<td>$S_2$</td>
</tr>
<tr>
<td>0</td>
<td>$S_1$</td>
<td>$S_3$</td>
<td>$S_4$</td>
</tr>
<tr>
<td>1</td>
<td>$S_2$</td>
<td>$S_5$</td>
<td>$S_6$</td>
</tr>
<tr>
<td>00</td>
<td>$S_3$</td>
<td>$S'_7$</td>
<td>$S'_7$</td>
</tr>
<tr>
<td>01</td>
<td>$S_4$</td>
<td>$S'_7$</td>
<td>$S'_{10}$</td>
</tr>
<tr>
<td>10</td>
<td>$S_5$</td>
<td>$S'_7$</td>
<td>$S'_{10}$</td>
</tr>
<tr>
<td>11</td>
<td>$S_6$</td>
<td>$S'_7$</td>
<td>$S'_7$</td>
</tr>
<tr>
<td>not (011 or 101)</td>
<td>$S'_7$</td>
<td>$S_0$</td>
<td>$S_0$</td>
</tr>
<tr>
<td>011 or 101</td>
<td>$S'_{10}$</td>
<td>$S_0$</td>
<td>$S_0$</td>
</tr>
</tbody>
</table>

Final Reduced State Machine

15 states (min 4 FFs) reduced to 7 states (min 3 FFs)
More Complex State Minimization

**Multiple input example**

![Diagram of state transition]  
 inputs here

<table>
<thead>
<tr>
<th>present state</th>
<th>next state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S0 S1 S2 S3</td>
<td>1</td>
</tr>
<tr>
<td>S1</td>
<td>S0 S3 S1 S4</td>
<td>0</td>
</tr>
<tr>
<td>S2</td>
<td>S1 S3 S2 S4</td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>S1 S0 S4 S5</td>
<td>0</td>
</tr>
<tr>
<td>S4</td>
<td>S0 S1 S2 S5</td>
<td>1</td>
</tr>
<tr>
<td>S5</td>
<td>S1 S4 S0 S5</td>
<td>0</td>
</tr>
</tbody>
</table>

symbolic state transition table

Minimized FSM

**Implication Chart Method**

- Cross out incompatible states based on outputs
- Then cross out more cells if indexed chart entries are already crossed out

![Implication chart diagram]  
 minimized state table (S0==S4) (S3==S5)
Minimizing Incompletely Specified FSMs

- Equivalence of states is transitive when machine is fully specified
- But it's not transitive when don't cares are present
  
<table>
<thead>
<tr>
<th>state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>-0</td>
</tr>
<tr>
<td>S1</td>
<td>1-</td>
</tr>
<tr>
<td>S2</td>
<td>-1</td>
</tr>
</tbody>
</table>

  S1 is compatible with both S0 and S2
  But S0 and S2 are incompatible

- No polynomial time algorithm exists for determining best grouping of states into equivalent sets that will yield the smallest number of final states

Example: edge detector - outputs 1 when last two input changes from 0 to 1

\[
\begin{array}{c|c|c|c|c}
X & Q_1 & Q_0 & Q_1^+ & Q_0^+ \\
--- & --- & --- & --- & --- \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
- & 1 & 0 & 0 & 0 \\
\end{array}
\]

\[Q_1^+ = X \ (Q_1 \text{ xor } Q_0)\]
\[Q_0^+ = X \ Q_1' \bar{Q}_0'\]
Another Implementation of Edge Detector

- "Ad hoc" solution - not minimal but cheap and fast

State Assignment

- Choose bit vectors to assign to each "symbolic" state
  - With \( n \) state bits for \( m \) states there are \( 2^n! / (2^n - m)! \)
    - \( \log n \leq m \leq 2^n \)
  - \( 2^n \) codes possible for 1st state, \( 2^n-1 \) for 2nd, \( 2^n-2 \) for 3rd, ...
  - Huge number even for small values of \( n \) and \( m \)
    - Intractable for state machines of any size
    - Heuristics are necessary for practical solutions
  - Optimize some metric for the combinational logic
    - Size (amount of logic and number of FFs)
    - Speed (depth of logic and fanout)
    - Dependencies (decomposition)
State Assignment Strategies

- Possible Strategies
  - Sequential - just number states as they appear in the state table
  - Random - pick random codes
  - One-hot - use as many state bits as there are states (bit=1 -> state)
  - Output - use outputs to help encode states
  - Heuristic - rules of thumb that seem to work in most cases

- No guarantee of optimality - another intractable problem

One-hot State Assignment

- Simple
  - Easy to encode, debug

- Small Logic Functions
  - Each state function requires only predecessor state bits as input

- Good for Programmable Devices
  - Lots of flip-flops readily available
  - Simple functions with small support (signals its dependent upon)

- Impractical for Large Machines
  - Too many states require too many flip-flops
  - Decompose FSMs into smaller pieces that can be one-hot encoded

- Many Slight Variations to One-hot
  - One-hot + all-0
State Maps and Counting Bit Changes

Bit Change Heuristic

```
S0 -> S1: 2 1
S0 -> S2: 3 1
S1 -> S3: 3 1
S2 -> S3: 2 1
S3 -> S4: 1 1
S4 -> S1: 2 2
Total: 13 7
```

Adjacency Heuristics for State Assignment

- Adjacent codes to states that share a common next state
  - Group 1’s in next state map
    
    \[
    \begin{array}{ccc}
    i & Q & Q^+ \\
    i & a & c \\
    i & b & c \\
    \end{array}
    \]
    
    \[
    c = i \cdot a + i \cdot b
    \]

- Adjacent codes to states that share a common ancestor state
  - Group 1’s in next state map
    
    \[
    \begin{array}{ccc}
    i & Q & Q^+ \\
    i & a & b \\
    k & a & c \\
    \end{array}
    \]
    
    \[
    b = i \cdot a \\
    c = k \cdot a
    \]

- Adjacent codes to states that have a common output behavior
  - Group 1’s in output map
    
    \[
    \begin{array}{ccc}
    i & Q & Q^+ \\
    i & a & b \\
    i & c & d \\
    \end{array}
    \]
    
    \[
    j = i \cdot a + i \cdot c \\
    b = i \cdot a \\
    d = i \cdot c
    \]
Heuristics for State Assignment

Successor/Predecessor Heuristics

- **High Priority**: $S'_3$ and $S'_4$ share common successor state ($S_0$)
- **Medium Priority**: $S'_3$ and $S'_4$ share common predecessor state ($S'_1$)
- **Low Priority**:
  - 0/0: $S_0$, $S'_1$, $S'_3$
  - 1/0: $S_0$, $S'_1$, $S'_3$, $S'_4$

Heuristics for State Assignment

(a) First state assignment

(b) Second state assignment
Another Example

High Priority:
- $S'_3$, $S'_4$
- $S'_7$, $S'_10$

Medium Priority:
- $S_1$, $S_2$
- $2 \times S'_3$, $S'_4$
- $S'_7$, $S'_10$

Low Priority:
- $0/0$: $S_0$, $S_1$, $S_2$, $S'_3$, $S'_4$, $S'_7$
- $1/0$: $S_0$, $S_1$, $S_2$, $S'_3$, $S'_4$, $S'_7$, $S_{10}$

Example Continued

- Choose assignment for $S_0 = 000$
- Place the high priority adjacency state pairs into the State Map
- Repeat for the medium adjacency pairs
- Repeat for any left over states, using the low priority scheme

Two alternative assignments at the left
Why Do These Heuristics Work?

- Attempt to maximize adjacent groupings of 1's in the next state and output functions

<table>
<thead>
<tr>
<th>Current State</th>
<th>Next State</th>
<th>X = 0</th>
<th>X = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S₀) 000</td>
<td>001</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>(S₁) 001</td>
<td>011</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>(S₂) 101</td>
<td>010</td>
<td>010</td>
<td></td>
</tr>
<tr>
<td>(S₃) 110</td>
<td>010</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>(S₄) 100</td>
<td>000</td>
<td>000</td>
<td></td>
</tr>
</tbody>
</table>

General Approach to Heuristic State Assignment

- All current methods are variants of this
  1) Determine which states "attract" each other (weighted pairs)
  2) Generate constraints on codes (which should be in same cube)
  3) Place codes on Boolean cube so as to maximize constraints satisfied (weighted sum)

Different weights make sense depending on whether we are optimizing for two-level or multi-level forms

Can’t consider all possible embeddings of state clusters in Boolean cube

- Heuristics for ordering embedding
- To prune search for best embedding
- Expand cube (more state bits) to satisfy more constraints

![Diagram](image-url)
### Output-Based Encoding

- Reuse outputs as state bits - use outputs to help distinguish states
  - Why create new functions for state bits when output can serve as well
  - Fits in nicely with synchronous Mealy implementations

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Present State</th>
<th>Next State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>TL</td>
<td>TS</td>
<td>ST</td>
</tr>
<tr>
<td>0</td>
<td>–</td>
<td>–</td>
<td>HG</td>
</tr>
<tr>
<td>–</td>
<td>0</td>
<td>–</td>
<td>HG</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>–</td>
<td>HG</td>
</tr>
<tr>
<td>–</td>
<td>–</td>
<td>0</td>
<td>HY</td>
</tr>
<tr>
<td>–</td>
<td>–</td>
<td>1</td>
<td>HY</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>–</td>
<td>FG</td>
</tr>
<tr>
<td>0</td>
<td>–</td>
<td>–</td>
<td>FG</td>
</tr>
<tr>
<td>–</td>
<td>1</td>
<td>–</td>
<td>FG</td>
</tr>
<tr>
<td>–</td>
<td>–</td>
<td>0</td>
<td>FY</td>
</tr>
<tr>
<td>–</td>
<td>–</td>
<td>1</td>
<td>FY</td>
</tr>
</tbody>
</table>

HG = ST’ H1’ H0’ F1 F0’ + ST H1 H0’ F1’ F0
HY = ST H1’ H0’ F1 F0’ + ST’ H1’ H0 F1 F0’
FG = ST H1 H0 F1 F0’ + ST’ H1 H0’ F1’ F0’
HY = ST H1 H0’ F1’ F0’ + ST’ H1 H0’ F1’ F0

Output patterns are unique to states, we do not need ANY state bits - implement 5 functions (one for each output) instead of 7 (outputs plus 2 state bits)

### Current State Assignment Approaches

- For tight encodings using close to the minimum number of state bits
  - Best of 10 random seems to be adequate (averages as well as heuristics)
  - Heuristic approaches are not even close to optimality
  - Used in custom chip design

- One-hot encoding
  - Easy for small state machines
  - Generates small equations with easy to estimate complexity
  - Common in FPGAs and other programmable logic

- Output-based encoding
  - Ad hoc - no tools
  - Most common approach taken by human designers
  - Yields very small circuits for most FSMs
Sequential Logic Implementation Summary

- Implementation of sequential logic
  - State minimization
  - State assignment

- Implications for programmable logic devices
  - When logic is expensive and FFs are scarce, optimization is highly desirable (e.g., gate logic, PLAs, etc.)
  - In Xilinx devices, logic is bountiful (4 and 5 variable TTs) and FFs are many (2 per CLB), so optimization is not so crucial an issue as in other forms of programmable logic
  - This makes sparse encodings like One-Hot worth considering

This makes sparse encodings like One-Hot worth considering